

깨끗한 코드

2014년 8월 15일 박재호(jrogue@gmail.com)

*들어가기 앞서

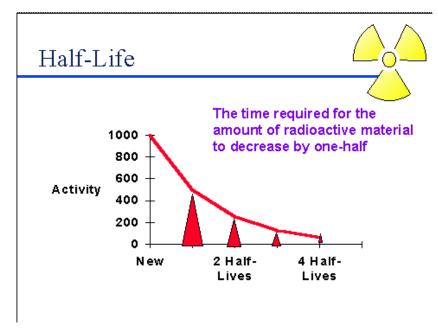
- 소프트웨어 프로페셔널이란?
- 프로페셔널은 어떻게 행동해야 할까?
- 프로페셔널은 충돌, 빠듯한 일정, 엄격한 관리자를 어떻게 다룰까?
- 언제, 어떻게 프로페셔널은 '아니오'라고 단호하게 거절해야 하나?
- 프로페셔널은 압력을 어떻게 다룰까?

*미켈란젤로와 시스티나 성당 천장벽화





+소프트웨어 코드 반감기





+ 원대한 재설계의 꿈



+ 깨진 창 이론



*코드 리펙터링의 목표

■ 깨끗한 코드를 유지한다!

* 가독성의 기본 정리

- 코드는 이해하기 쉬워야 한다.
- 코드는 **다른 사람**이 그것을 이해하는 데 들이는 시간을 최소화하는 방식으로 작성되어야 한다.
- 1회용 코드는 되도록 피해야 한다(스스로가 희생양이 될지도).
 - 참고: Perl 코드(WORN: Write Once Read Never)

*깨끗한 코드란?

- 비야네 스트롭스트룹(C++ 창시자)
 - 나는 우아하고 효율적인 코드를 좋아한다. 논리가 간단해야 버그가 숨어들지 못한다. 의존성을 최대한 줄여야 유지보수가 쉬워진다. 오류는 명백한 전략에 의거해 철저히 처리한다. 성능을 최적으로 유지해야 사람들이 원칙 없는 최적화로 코드를 망치려는 유혹에 빠지지 않는다. <u>깨끗한 코드는 한 가지를 제대로 한다.</u>



⁺깨끗한 코드란?

- 그레디 부치(객체지향 대가)
 - 깨끗한 코드는 단순하고 직접적이다. <u>깨끗한 코드는 잘 쓴 문장처럼</u> <u>읽힌다.</u> 깨끗한 코드는 결코 설계자의 의도를 숨기지 않는다. 오히려 명쾌한 추상화와 단순한 제어문으로 가득하다.



* 깨끗한 코드란?

- 데이브 토마스(실용주의 프로그래머)
 - 깨끗한 코드는 작성자가 아닌 사람도 읽기 쉽고 고치기 쉽다. 단위 테스트 케이스와 수용 테스트 케이스가 존재한다. 이런 이름은 의미가 있다. 특정 목적을 달성하는 방법은 (여러 가지가 아니라) 하나만 제공한다. 의존성은 최소이며 각 의존성을 명확히 정의한다. API는 명확하며 최소로 줄였다. 때로는 필요한 정보 전부를 코드만으로 명확하게 드러내기 어려우므로 언어에 따라 문학적 표현도 필요하다.



*깨끗한 코드란?

- 마이클 페더즈("Working Effectively with Legacy Code" 저자)
 - 깨끗한 코드의 특징은 많지만 그 중에서도 모두를 아우르는 특징이하나 있다. 깨끗한 코드는 언제나 누군가 주의 깊게 짰다는 느낌을 준다. 고치려고 살펴봐도 딱히 손 댈 곳이 없다. 작성자가 이미 모든 사항을 고려했으므로. 고칠 궁리를 하다보면 언제나 제자리로 돌아온다. 그리고는 누군가 남겨준 코드, 누군가 주의 깊게 짜놓은 작품에 감사를 느낀다.



҆ 깨끗한 코드란?

- 론 제프리("Extreme Programming Installed" 저자)
 - 최근 들어 나는 [켄트] 벡이 제안한 간단한 코드 규칙으로 구현을 시작한다. (그리고 같은 규칙으로 구현을 거의 끝낸다.) 중요한 순으로 나열하자면 간단한 코드는
 - 모든 테스트를 통과한다.
 - 중복이 없다.
 - 시스템 내 모든 설계 아이디어를 표현한다.
 - 클래스, 메소드, 함수 등을 최대한 줄인다.
 - 물론 나는 주로 중복에 집중한다. 같은 작업을 여러 차례 반복한다면 코드가 아이디어를 제대로 표현하지 못한다는 증거다. 나는 문제의 아이디어를 찾아내 좀더 명확하게 표현하려 애쓴다.
 - <u>중복 줄이기, 표현력 높이기, 초반부터 간단한 추상화 고려하기</u>. 내게는 이 세 가지가 깨끗한 코드를 만드는 비결이다.

⁺ 깨끗한 코드란?

- 워드 커닝엄(위키 창시자, 피트 창시자, 익스트림 프로그래밍 창시자)
 - 코드를 읽으면서 짐작했던 기능을 각 루틴이 그대로 수행한다면 깨끗한 코드라 불러도 되겠다. 코드가 그 문제를 풀기 위한 언어처럼 보인다면 아름다운 코드라 불러도 되겠다.



* 코드 읽기 vs 코드 쓰기

- 읽기: 쓰기 = <u>10: 1</u>
 - 비율이 이렇게 높으므로 읽기 쉬운 코드가 매우 중요하다. 비록 읽기 쉬운 코드를 짜기가 쉽지는 않더라도 말이다. 하지만 기존 코드를 읽어야 새 코드를 짜므로 읽기 쉽게 만들면 사실은 짜기도 쉬워진다.
 - 이 논리에서 빠져나갈 방법은 없다. 주변 코드를 읽지 않으면 새 코드를 짜지 못한다. 주변 코드를 읽기가 쉬우면 새 코드를 짜기도 쉽다. 주변 코드를 읽기가 어려우면 새 코드를 짜기도 어렵다. 그러므로 급하다면, 서둘러 끝내려면, 쉽게 짜려면, 읽기 쉽게 만들면된다.

* 현실과 비현실

- 데이비드 파나스(소프트웨어 공학 태두)
 - 소프트웨어 설계자가 요구 사항으로부터 이성적이고 오류가 없는 방식으로 설계한 그림은 엄청나게 비현실적입니다. 어느 시스템도 이런 이성적인 방식으로 개발되지 않았으며, 아마도 앞으로도 이런 일은 생기지 않을 것입니다. 심지어 교과서나 논문에 나온 작은 프로그램 개발조차도 비현실적입니다. 교과서나 논문에 나온 프로그램은 개선되고 수정되는 과정을 거치면서 저자들이 바라는 바를 보여주지 실제 일어난 과정을 보여주지는 않습니다.

+ 이름

이모지(emoji)를 이름으로 사용한다면?(swift)

```
class & & & &
 func & & & (5: Int, $3: Int) -> Int
  return 😇 + 🐰
var 😓 = 🐔 + 2
var @ = @ @ @ @ ()
```

+ 함수(1)

수학 함수라고 해서 어려울 필요는 없다.

```
/* Newton-Raphson Method */
1  /* Newton-Raphson Method */
2  float rtsafe(void (*funcd)(float, float *, float *), float x1, float x2, float xacc)
3
```

힌트:

- i) 함수 포인터 타입 정의
- ii) 좌표를 구조체로

+ 함수(2)

표현을 잘개 쪼개기

- 거대한 표현을 작은 조각으로 나누고 함수로 분리한다.
- 하위표현을 간결한 이름으로 대체해 함수로 분리한다.
- 코드를 읽는 사람이 핵심 '개념'을 파악하게 읽기 쉬운 코드를 작성한다.

+ 변수

변수는 내버려두면 계속 늘어난다

- 변수를 덜 사용하고 최대한 '가볍게' 만들어 가독성을 높인다.
- 실천 방안
 - 방해되는 변수를 제거: 결과를 즉시 처리하는 방식으로, 중간 결과값을 저장하는 변수를 제거
 - 각 변수의 범위를 최대한 작게 줄임: 각 변수 위치를 옮겨 변수가 나타나는 줄의 수를 최소화. 안 보이면 멀어진다.
 - 값이 한번만 할당되는 변수를 선호: const, final 등으로 값이 한 번만 할당되어 바뀌지 않는(즉 불변) 변수는 이해하기 쉽다.

*코드 흐름 제어

코드 흐름을 읽기 쉽게 만들어야 논리가 명확해진다

- 실천 방안
 - if/else문의 블록 순서에 주의: 긍정적이고 쉽고 흥미로운 경우가 앞쪽에 위치
 - 삼항연산자(: ?)나 do/while, goto는 코드 가독성을 떨어뜨리므로 되도록 사용 금지
 - 중첩된 코드 블록의 흐름을 따라가려면 많은 집중이 필요하다 → 선형적인 코드 추구
 - 함수 중간에 반환하면 중첩을 피하고 코드를 더 깔끔하게 작성 가능하다. → 함수 앞부분에서 '보호 구문'으로 간단한 경우를 미리 처리하는 방안도 고려

* 주석

잡음을 줄이고 의도를 명확하게

```
/* noise */

1  /* noise */
2
3  /** The name. */
4  static String name;
5  /** The version. */
6  static String version;
7  /** The licenceName. */
8  static String licenceName;
9  /** The version. */
10  static String info;
11
```

```
/* intention */

1 /* intention */

2 
3 // 직원에게 복지 혜택을 받을 자격이 있는지 검사한다.
4 if ((employee.flags & HOURLY_FLAG) &&
5 (employee.age > 65))

6 
7 /* vs */

8 
9 if (employee.isEligibleForFullBenefits())
```

* 주석(2)

잡음을 줄이고 의도를 명확하게

- 설명하지 말하야 할 내용
 - 코드 자체에서 재빨리 도출될 수 있는 사실
 - 나쁜 함수명과 같이 나쁘게 작성된 코드를 보정하려 '애쓰는 주석' → 코드 수정!
- 기록해야 하는 생각
 - 코드가 특정 방식으로 작성된 이유를 설명하는 내용
 - 코드에 담긴 결함 → TODO:, XXX: 와 같은 표기법 사용(IDE 인식)
 - 어떤 상수가 특정 값으로 정해진 이유
- ■기타
 - 평범한 사람이 예상치 못한 특이한 동작을 기록
 - 파일이나 클래스 수준 주석에서 '큰 그림'을 설명하고 각 조각이 어떻게 맞춰지는지를 설명

+ 형식 맞추기(1)

파일 길이는 짧아야 한다(유명 오픈소스 코드 통계)

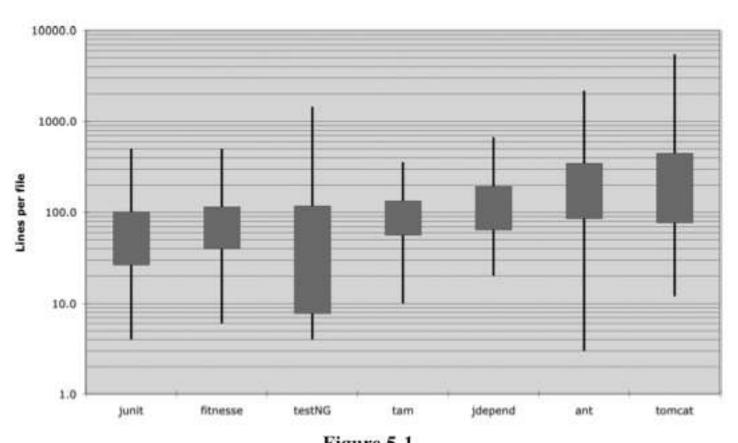


Figure 5-1
File length distributions LOG scale (box height = sigma)

*형식 맞추기(2)

행 길이도 짧아야 한다(유명 오픈소스 코드 통계)

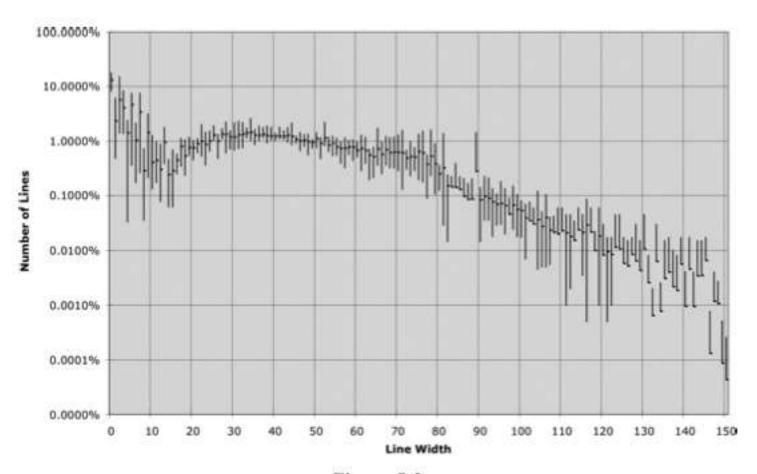


Figure 5-2
Java line width distribution

* 형식 맞추기(3)

미학적인 요소 고려

- 여러 블록에 담긴 코드가 모두 비슷한 일을 하면 실루엣이 동일해 보이게 구성
- 코드 곳곳을 '열'로 만들어 줄을 맞추면 코드를 한 눈에 훑어보기 편하게 구성
- 코드 한 곳에서 A, B, C 순으로 언급되면, 다른 곳에서 B, A, C 순으로 언급하지 마라. 의미있는 순서를 정해 지켜라.
- 빈줄을 이용해 커다란 블록을 논리적인 '문단'으로 나눠라.

* 코드 분량 줄이기

가급적이면 적은 코드를 유지

- 가장 읽기 쉬운 코드는 아무것도 없는 코드!
- 테스트/유지보수/문서화에 유리
- 실천 방안
 - 제품에 필요하지 않는 기능은 제거(주석 처리 금물!)
 - 요구 사항을 재고해서 가장 단순한 형태의 문제를 탐색
 - 주기적으로 프레임워크/라이브러리의 전체 클래스/API를 훑어보고 표준으로 제공하는 기능에 친숙해짐

* 절차적인 코드와 객체 지향 코드

객체 지향이 만능인가?

- 객체는 동작을 공개하고 자료를 숨긴다. 그래서 기존 동작을 변경하지 않으면서 새 객체 타입을 추가하기는 쉬운 반면, 기존 객체에 새동작을 추가하기는 어렵다. 자료 구조는 별다른 동작 없이 자료를 노출한다. 그래서 기존 자료 구조에 새 동작을 추가하기는 쉬우나, 기존 함수에 새 자료 구조를 추가하기는 어렵다.
- 절차적인 코드는 새로운 자료 구조를 추가하기 어렵다. 그러려면 모든 함수를 고쳐야 한다. 객체 지향 코드는 새로운 함수를 추가하기 어렵다. 그러려면 모든 클래스를 고쳐야 한다.
- (어떤) 시스템을 구현할 때, 새로운 자료 타입을 추가하는 유연성이 필요하면 객체가 더 적합하다. 다른 경우로 새로운 동작을 추가하는 유연성이 필요하면 자료 구조와 절차적인 코드가 더 적합하다. 우수 한 소프트웨어 개발자는 편견 없이 이 사실을 이해해 직면한 문제에 최적인 해결책을 선택한다.

* 오류 처리

null 반환과 전달

- null의 의미는 두 가지
 - 정말 없는 것
 - 없다고 표현한 것
- 모호성으로 인한 문제점은 셀 수 없다
 - null을 반환하거나 null을 전달하지 마라!
 - 정말 null이 필요하면 optional을 고려(참고: C++에서는 boost에 포함)

```
package jrogue
import com.google.common.base.Optional;

public class BasicTest {
    @Test
    public void optionalAbsentTest() throws Exception {
        Optional absent = Optional.absent();
        System.out.println("Is absent present? " + absent.isPresent());

}

System.out.println("absent is " + absent.get());
}
```

+ 테스트

가독성이 가장 중요한 위치를 차지

- 테스트가 읽기 편해야 유지보수/확장도 쉬워진다.
- 실천 방안
 - 각 테스트의 최상위 수준은 최대한 간결하게 유지
 - 테스트에 실패하면, 버그를 추적해 수정하게 돕는 오류 메시지를 출력해야 함
 - 코드의 구석구석을 철저하게 실행하는 가장 간단한 입력을 사용
 - 무엇이 테스트되는지 분명하게 드러나게 테스트 함수에 충분한 설명이 포함된 이름을 부여
 - 무엇보다 테스트의 수정이나 추가가 쉬워야 한다!

* 단위 테스트

F.I.R.S.T.

- 빠르게(Fast): 테스트는 빨라야 한다.
- 독립적으로(Independent): 각 테스트는 서로 의존하면 안 된다. 한 테스트가 다음 테스트가 실행될 환경을 준비해서는 안 된다.
- 반복가능하게(Repeatable): 테스트는 어떤 환경에서도 반복 가능해야 한다.
- 자가검증하는(Self-Validating): 테스트는 부울(bool) 값으로 결과를 내야 한다. 성공 아니면 실패다.
- 적시에(Timely): 테스트는 적시에 작성해야 한다. 단위 테스트는 테스트하려는 실제 코드를 구현하기 직전에 구현한다.



+클래스

SRP, Single Responsibility Principle

```
7+ class +/
public class SuperDashbeard extends JFrame implements Metabatabler
  public String getCustomizerLanguagePath()
public void setSystemConfigNath(String systemConfigNath)
public String getSystemConfigNath(String systemConfigNath)
public void setSystemConfigNateman(String systemConfigNatemant)
    public boolean getGuruState()
    public beolean getHaviceState()
    public Boolean getOpenSourceState()
    public word showObject(MetaObject object)
public void sharProgress[String s]
public bolico: isMerjoda/shirty!
public void setisMerjoda/shirty!bolicon isMerjodataDirty!
public Emparent getLantFouredComposent!
public void setLantFouredComposent!
    public void setMouseSelectStatelboolean isMouseSelected)
    public boolean isMouseSelected!
    public LanguageManager getLenguageManager()
    public Project getProject()
    public Project getFiritFrojecc()
    public Project getLastProject()
    public String getMen#rojectName()
public word setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public wood setCurrentDiriString newBirl
    public void updateStatustint dotPos, int marmPosl
    public Class[] gotDataBaseClasses[]
    public MetadataFeeder getMetadataFeederil
    public word addProjectiProject project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
    ponic Metarojectikade gethrogrambetadatni)
ponic void resetbaskourdi
ponic Project LoodFrojectiString tileName, String projectName)
ponic void sectorSureMetadatnilaeOpen cardowe)
    public Metalbject getSelectedObject()
    public word setProject(Project project)
    public word editorAction(String actionName, ActionEvent event)
    public word setMode(int mode)
    public fileflesager getfileflesager()
public void setfileflesager(fileflesager fileflesager)
    public ConfligHasager getConfligHasager()
    public word notConFigNanager(ConFigNanager conFigNanager)
    public ClassCoater getClassCoater)
    public word setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome(
    public String getBaseDirt)
    public int getMajorVersierMumber()
    public int getMinorVersionNumber()
public Metablect pasting()
Metablect target, Metablect pasted, MetaProject project)
public void procesyMenuSeparators(Metablect metablect)
public void procesyMenuSeparators(Metablect metablect)
public void processTabFages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
    public wood updateOisplayLayer(MetaGhject object, int layerIndex)
    public would propertyEditedSepaint/NetaObject object)
    public void procesiBeletsObject(MetaObject skjert)
public booleen getättachedToDesigner()
   public upid processProjectChangedState[bgelwas hasProjectChanged]
public wold processBbjectNaseChangedPmtaDbject object;
public void setAcoubragging(boolean atlaubragging)
    public boolean allowbragging()
    public boolean isCustomizing()
    public wold setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showledger(MetaObject metaObject, String propertyName)
// ... 日日 明刊月 时企业月 中の日日 ...
```

```
/* class */
     /* class */
     public class SuperDashboard extends JFrame implements MetaDataUser
         public Component getLastFocusedComponent()
         public void setLastFocused(Component lastFocused)
         public int getMajorVersionNumber()
         public int getMinorVersionNumber()
         public int getBuildNumber()
10
```

*시스템

생성과 사용의 분리

- 팩토리
- 의존성 주입(DI)과 제어 역전(IoC)
 - 예: 스프링 프레임워크
- 생각해볼 문제
 - "처음부터 올바르게" 시스템을 만들 수 있다는 믿음은 <u>미신</u>이다. 대신에 우리는 오늘 주어진 사용자 스토리에 맞춰 시스템을 구현해야 한다.
 - <u>소프트웨어 부문에서 가장 큰 악성 재고는 개발자의 낭비되는 시간</u>이다.
 - 소프트웨어 시스템은 물리적인 시스템과 다르다. <u>관심사</u>를 적절히 분리해 관리한다면 소프트웨어 아키텍처는 점진적으로 발전할 수 있다.
 - 소프트웨어 시스템은 "<u>수명이 짧다</u>"는 본질로 인해 아키텍처의 점진적인 발전이 가능하다.

+ 설계

켄트 백의 간단한 설계 규칙 네 가지

- 1. 모든 테스트를 실행한다.
- 2. 중복을 없앤다.
- 3. 프로그래머 의도를 표현한다.
- 4. 클래스와 메소드 수를 최소로 줄인다.

* 동시성(1)

동시성 개괄

- 동시성의 의의
 - 동시성은 결합(coupling)을 없애는 전략이다.
 - 즉, *무엇(what)*과 *언제(when)*을 분리하는 전략이다.
- 동시성의 장점
 - *무엇*과 *언제*를 분리하면 응용 프로그램 구조와 효율이 극적으로 나아진다. 구조적인 관점에서 프로그램은 거대한 루프 하나가 아니라 작은 협력 프로그램 여럿으로 보인다. 따라서 시스템을 이해하기가 쉬워지며 문제를 분리하기도 쉬워진다.

* 동시성(2)

동시성의 오해와 문제점

- 동시성의 오해
 - 동시성은 항상 성능을 높여준다.
 - 동시성을 구현해도 설계는 변하지 않는다.
 - 적절한 도구를 사용하면 동시성을 이해할 필요가 없다.
- 동시성의 문제점
 - *동시성은 다소 부하를 유발한다*. 성능 측면에서 부하가 걸리며, 코드도 더 짜야 한다.
 - *동시성은 복잡하다.* 간단한 문제라도 동시성은 복잡하다.
 - *일반적으로 동시성 버그는 재현하기 어렵다.* 그래서 일회성 문제로 여겨 무시하기 쉽다. 진짜 결함으로 간주되지 않는다.
 - 동시성을 구현하려면 흔히 근본적인 설계 전략을 재고해야 한다.

* 동시성(3)

구체적인 지침

- 구체적인 지침
 - 말이 안 되는 실패는 잠정적인 스레드 문제로 취급하라.
 - 다중 스레드를 고려하지 않은 순차 코드부터 제대로 돌게 만들자.
 - 다중 스레드를 쓰는 코드 부분을 쉽게 다양한 환경에 끼워 넣을 수 있게 스레드 코드를 구현하라.
 - 다중 스레드를 쓰는 코드 부분을 상황에 맞춰 조정할 수 있게 작성하라.
 - 프로세서 수보다 많은 스레드를 돌려보라.
 - 다른 플랫폼에서 돌려보라.
 - 코드에 보조 코드(instrument)를 넣어 돌려라. 강제로 실패를 일으키게 해보라.

* 점진적인 개선

방망이 깎던 노인

- 프로그래밍은 과학보다 공예에 가깝다.
 - 퇴고가 없는 작문은 존재하지 않는다. 즉, 깨끗한 코드를 짜려면 먼저 지저분한 코드를 짠 *뒤에 정리해야 한다*는 의미다.
- 자살 행위: 초보자는 일단 프로그램이 "돌아가면" 다음 업무로 넘어간다. "돌아가는" 프로그램은 그 상태가 어떻든 그대로 버려둔다. 경험이 풍부한 전문 프로그래머라면 이런 행동이 전문가로서 자살 행위라는 사실을 잘 안다.
 - 코드에 만족하는 프로그래머는 전문가 정신이 부족하다. 나쁜 코드보다 더 오랫동안 더 심각하게 개발 프로젝트에 악영향을 미치는 요인도 없다. 나쁜 일정은 다시 짜면 된다. 나쁜 요구사항은 다시 정의하면 된다. 나쁜 팀 역학은 복구하면 된다. 하지만 나쁜 코드는 썩어 문드러진다.
- 1년 전 코드보다 5분 전 코드가 정리하기가 편하다!

*보이스카우트 규칙

- 캠프장은 처음 왔을 때보다 더 깨끗하게 해놓고 떠나라.
- 전문가 정신은 지속적인 개선을 의미한다.

*참고 문헌

